



Informàtica

ICB0	Desenvolupament d'aplicacions multiplataforma
ICC0	Desenvolupament d'aplicacions web
M03	Programació
UF6	POO. Introducció a la persistència en BD Accés a BDR-BDOR-BDOO
Diari d'activitats	
Isidre Guixà	
Curs 2024/2025	



Connectivitat via JDBC – [Vídeo 01](#)

17/09/24
19/09/24

JDBC és un estàndard de connexió que incorpora Java per connectar amb els SGBDR.

Per poder-lo utilitzar per connectar amb un SGBDR, cal que el fabricant del SGBDR ens faciliti el connector JDBC adequat, que normalment es pot descarregar dels llocs web dels fabricants de SGBDR.

Java incorpora un munt d'interfícies (no classes) en el paquet `java.sql` que nosaltres utilitzarem en els programes que necessitin connexió amb un SGBDR.

- Per compilar els programes, NO necessitem cap driver del fabricant, doncs només necessitem la definició de les interfícies JDBC, que venen incorporades en Java.
- Per executar un programa, el projecte ha de tenir incorporat el connector facilitat pel fabricant del SGBDR i ha de ser coherent amb la versió de Java i la versió del SGBDR.

Informació dels drivers i dades necessàries per connectar amb els "nostres SGBDR habituals":

- Dades per connectar amb Oracle (port habitual: 1521)
Driver: `ojdbc8_18.3.jar` (per Oracle 18 - descarregable de l'aula del Classroom)
Driver: `ojdbc8_21.3.jar` (per Oracle 21 - descarregable de l'aula del Classroom)
URL: `jdbc:oracle:thin:@//IP:PORT/nomBD`
Classe JDBC: `oracle.jdbc.driver.OracleDriver`

- Dades per connectar amb PostgreSQL (port habitual: 5432)
Driver: `postgresql-42.2.5.jar` (descarregable de l'aula del Classroom)
URL: `jdbc:postgresql://IP:PORT/nomBD`
Classe JDBC: `org.postgresql.Driver`

- Dades per connectar amb MariaDB (port habitual: 3306)
Driver: `mariadb-java-client-3.3.2` (aula del Classroom)
URL: `jdbc:mariadb://IP:PORT/nomBD`
Classe JDBC: `org.mariadb.jdbc.Driver`

- Dades per connectar amb MySQL8 (port habitual: 3306)
Driver: `mysql-connector-java-8.0.13` (descarregable de l'aula del Classroom)
URL: `jdbc:mysql://IP:PORT/nomBD`
Classe JDBC: `com.mysql.cj.jdbc.Driver`

Si apareix error `java.sql.SQLException: The server timezone value ... is unrecognized or represents more than one timezone` cal utilitzar:

URL: `jdbc:mysql://IP:PORT/nomBD?serverTimezone=UTC`

- Dades per connectar amb MySQL5 (port habitual: 3306)
Driver: `mysql-connector-java-5.1.41-bin_JDJ8-7-6` (aula del Classroom)
URL: `jdbc:mysql://IP:PORT/nomBD`
Classe JDBC: `com.mysql.jdbc.Driver`

- Dades per connectar via ODBC (abans Java8):
Driver: Incorporat en Java (desapareix en Java8)
URL: `jdbc:odbc:NomDSN`
Classe JDBC: `sun.jdbc.odbc.JdbcOdbcDriver`

Pel cas d'ODBC (obsolet) caldrà que la màquina tingui instal·lat el connector ODBC adequat al SGBD a connectar i, en el cas de Windows, cal tenir en compte que distingeix entre connectors ODBC de 32 bits i connectors ODBC de 64 bits.

El protocol JDBC ha anat evolucionant junt amb l'evolució de JAVA.

- En Java6 va aparèixer JDBC4 que incorpora un mecanisme que fa fàcil establir connexió amb un SGBDR.
- La versió estable actual és JDBC4.3



En cas d'haver de fer/retocar un programa en versió Java anterior a JDK6, cal tenir en compte que abans d'intentar establir connexió, cal carregar la classe corresponent al connector del fabricant, amb la instrucció:

```
Class.forName(nomDeLaClasseDelConnectorDelFabricant)
```

Però el fet d'estar en Java6 o posterior NO implica que ja no calgui carregar la classe via `Class.forName`, doncs dependrà també de si el connector JDBC és 4.x o anterior. Manera simple d'esbrinar-ho és mirar si el connector conté el fitxer `META-INF/services/java.sql.Driver` amb el nom de la classe a carregar en el seu interior.

Informació relativa als projectes de NetBeans que s'adjunten a aquest material:

- Tots estan configurats per a JDK17, que és la versió oficial de Java que utilitzem a l'escola en aquest curs. Tots poden funcionar en JDK8+, però caldrà retocar la configuració del projecte.
- Tots ells, a l'hora d'establir la connexió, incorporen la IP, port, usuari i contrasenya de la màquina on el professor té el SGBD. Cal que l'alumne les canviï per les dades que corresponguin.
- Les dades de connexió (url, usuari, contrasenya) no haurien d'estar mai incorporades en el codi, sinó que cal ubicar-les en fitxer de configuració (en Java s'acostuma a usar un fitxer `properties`).
- Recomanable tenir en el SGBD un esquema/usuari específic per la UF (per exemple, `m03uf6`) per no barrejar amb altres UFs.
- Els exemples usen taules en Oracle generades pel guió `empresaOracleMR.sql` ubicat a Classroom.
- Un projecte que hagi de connectar amb un SGBDR necessita incorporar en el projecte el corresponent connector, que pot residir a qualsevol disc/unitat de la màquina. Evidentment, es diferent la ubicació on el tindrà el professor que la ubicació on el tindrà l'alumne. Per a que no calgui anar a cercar cada vegada les ubicacions, els projectes utilitzen alguna/es de les llibreries:
 - JDBC Oracle, que apunta al connector per Oracle
 - JDBC PostgreSQL, que apunta al connector per PostgreSQL
 - JDBC MySQL, que apunta al connector per MySQL
 - JDBC MariaDB, que apunta al connector per MariaDB

Per tant, es recomana tenir aquestes llibreries configurades en el NetBeans i no caldrà retocar ubicacions.

- Projecte `240919_1_ConnexioJDBC_Oracle_SenseDriverJDBC` permet comprovar que el programa compila però, en executar-lo, es queixa que no troba un driver/connector per la URL indicada.
- Projecte `240919_2_ConnexioJDBC_Oracle_AmbDriverJDBC` permet comprovar que s'estableix la connexió si el driver/connector és instal·lat en el projecte i la URL és correcta (IP, port, usuari i contrasenya correctes) i el SGBD està engegat i hi ha connectivitat de xarxa.
- Projecte `240919_3_ConnexioJDBC_PostgreSQL_AmbDriverJDBC` permet comprovar que s'estableix la connexió si el driver/connector és instal·lat en el projecte i la URL és correcta (IP, port, usuari i contrasenya correctes) i el SGBD està engegat i hi ha connectivitat de xarxa i el servidor PostgreSQL està configurat per admetre connexions des de la màquina que vol connectar. Per defecte, els servidors PostgreSQL venen configurats per permetre connexions des de la mateixa màquina. Si és una altra... caldrà configurar-lo!!!

Establiment de connexió – Gestionar `autoCommit` – [Vídeo 1](#)

24/09/24

- Projecte `240924_1_ConnexioJDBC_ControlCommit`

Incorpora:

- Com esbrinar l'estat d'`AutoCommit`, que en JDBC sempre ve (en teoria) a `true`.
- Desactivar l'`AutoCommit`.

NO és adequat incorporar URL, usuari, contrasenya... dins el codi. És adequat usar fitxer de propietats.

- Projecte `240924_2_ConnexioJDBC_IndependentDelSGBD`

Les dades de la connexió passen a residir en un fitxer al què s'accedeix abans d'establir la connexió.

Podria ser un fitxer de text amb una sintaxis definida per nosaltres (per exemple, la URL a la primera línia, l'usuari a la segona línia,...) però per facilitar la gestió dels valors a recuperar, en Java s'acostuma a usar els fitxers de



propietats (text –habitualment amb extensió `properties`- o XML), ja que disposem de la classe `Properties` per a fer-ne una ràpida recuperació dels valors.

El projecte 240924_2 conté les dades de connexió (URL, usuari i contrasenya) en un fitxer de propietats anomenat `config.properties` (podria ser qualsevol nom) i, en posar en marxa el programa, via la classe `Properties` s'efectua la recuperació dels valors necessaris per establir la connexió.

El programa del projecte intenta recuperar del fitxer de propietats les propietats `url`, `user` i `pwd` en les variables `url`, `user` i `pwd` del programa per després usar-les per establir la connexió.

Recuperació d'informació (SELECT) via JDBC – [Vídeo 2](#)

26/09/24

- Sentència `createStatement` per crear un objecte `Statement` (sentència)
- A través de un objecte `Statement`, obtenir un objecte `ResultSet` via `executeQuery`
- Processar el `ResultSet`, recuperant els valors de cada columna via mètode `get` adequat al tipus de la columna.
- Accés possible a la columna via posició (comptant a partir d'1) o via nom de columna.
- **Alerta** amb l'actuació de `ResultSet.get...()` quan la columna a la BD conté valor `null`:
 - Alguns retornen `null`, com `getString(...)`
 - Altres, com els numèrics, retornen 0 i... evidentment un `null` no és igual a ZERO!!!

El programador ha de ser conscient d'aquest fet i, quan vulgui saber si una columna conté valor `null`, després d'efectuar la lectura (via `get`), disposa del mètode `ResultSet.isNull()` per esbrinar si la **darrera** lectura corresponia a valor `null`.

Exercici per dimarts, 1 d'octubre:

Finalitzar el Programa del projecte 240926_1_RecuperarDepartaments de manera que mostri els departaments de l'empresa en format:

CODI	DNOM	LOC
10	COMPTABILITAT	SEVILLA
20	INVESTIGACIÓ	MADRID
30	VENDES	BARCELONA
40	PRODUCCIÓ	BILBAO

JDBC – Compte amb valors Date i numèrics null

01/10/24

- JDBC gestiona els camps data amb la classe `java.sql.Date` enlloc de la classe `java.util.Date` i cal tenir present que `java.sql.Date` deriva de `java.util.Date`.

Exercici per lliurar a Classroom (tasca 1):

Projecte de nom `T1_Cognom1Cognom2Nom` amb programa que mostri la següent informació de tots els empleats:

CODI	COGNOM	DEPARTAMENT	CAP	DATA ALTA	OFICI	SALARI	COMISSIÓ
-----	-----	-----	-----	-----	-----	-----	-----

on:

- Columna `DEPARTAMENT` ha de mostrar el nom del departament
- Columna `CAP` ha de mostrar el cognom del cap
- Columna `DATA ALTA` ha de mostrar la data en format `dd/mm/yyyy`

Exercici per lliurar a Classroom (tasca 2): Projecte `T2_Cognom1Cognom2Nom`

Mostrar per la consola, la informació de totes les comandes, amb les seves línies, ordenades per número de comanda i les línies per número de línia, en format similar a:

Comanda:

```
Número: xxx
Data Comanda: xxx (format dd/mm/yyyy)
Tipus:
Data Tramesa: xxx (format dd/mm/yyyy)
Client: Codi - Nom
Import total: xxx
```



```
Línies:
    Número: xxx
        Producte: Codi - Descripció
        Qtat: xxxx - Preu: xxxx - Import: xxxx
    Número: xxx
        Producte: Codi - Descripció
        Qtat: xxxx - Preu: xxxx - Import: xxxx
    ...
Comanda: ...
```

Versió 1 a desenvolupar: Usar una única `SELECT` que recuperi comandes i línies.

Caldrà recórrer el `ResultSet` per poder mostrar les línies, però totes les línies d'una mateixa comanda contenen la informació de la comanda i aquesta només ha de sortir abans de la primera línia. Per tant, caldrà un semàfor per controlar quan es canvia de comanda i, en aquest cas, mostrar les dades de la comanda.

Tingueu present que podria donar-se el cas d'existir alguna comanda sense línies, i també ha d'aparèixer.

Versió 2 del programa anterior (tasca 3): Usar dues `SELECT`:

- `SELECT` que obtingui totes les comandes
- `SELECT` que obtingui les línies de detall de cada comanda

Cal fer un recorregut per les comandes obtingudes en la primera `SELECT` i per a cada comanda cal executar la segona `SELECT` per obtenir les línies de detall de la corresponent comanda.

Atenció:

- Per executar la primera `SELECT` necessitem un `Statement` (suposem `staCom`) pel que executarem la `SELECT` i obtindrem un `ResultSet` (suposem `rsCom`) que anirem recorrent.
- Donada una comanda, necessitem executar una altra `SELECT`, que precisa d'un `Statement`. **NO** es pot usar el mateix `staCom` de la primera `SELECT`, doncs provocaria el tancament del `ResultSet` `rsCom` i no podríem continuar recorrent `rsCom`.
És a dir, cada `ResultSet` amb el seu `Statement`, doncs tots 2 treballen simultàniament

Instruccions parametritzades per efectivitat i per evitar injecció de codi – [Vídeo](#)

10/10/24

En el vídeo s'explica com efectuar una consulta parametritzada i l'autor explica la importància d'utilitzar instruccions parametritzades per evitar injecció de codi. En realitat hi ha dos motius per usar sentències parametritzades:

- Millorar eficiència

Suposem una consulta per recuperar nom i sou dels empleats d'un departament indicat pel seu codi, que pot anar canviant i que té en una variable `dept` (suposem numèrica)

Un usuari inexpert podria pensar en dissenyar la consulta següent:

```
String s = "select nom, sou from emp where id_dept = " + dept; (1)
```

En cas que la variable `dept` fos cadena, caldria tancar-la entre cometes simples a la consulta:

```
s = "select nom, sou from emp where id_dept = '" + dept + "'"; (2)
```

Cada vegada que volgués obtenir la informació, executaria (suposant que `st` és `Statement`)

```
Query q = st.executeQuery(s);
```

Suposem que executa la consulta diverses vegades havent canviat el valor de `dept` per 10, 20, 30.

El SGBD hauria rebut les consultes:

```
select nom, sou from emp where id_dept = 10;
select nom, sou from emp where id_dept = 20;
select nom, sou from emp where id_dept = 30;
```

Quan un SGBD rep una instrucció, l'analitza, mira que sigui correcte i crea el pla d'execució (programa intern) per poder efectuar el que demana la instrucció. Per millorar eficiència, es guarda la instrucció i el pla



d'execució, de manera que cada vegada que arriba una instrucció, mira si la té guardada i si la té (ha d'estar exactament escrita) no cal que l'analitzi ni que creï el pla d'execució, per què ja ho he fet i pot passar directament a l'execució del pla.

Però en el cas anterior, **les tres instruccions que “fan el mateix”, no són idèntiques**, doncs canvia el valor 10, 20,... i ha de fer l'anàlisi i elaborar el pla d'execució cada vegada.

Treballar amb sentències parametritzades evita el problema, per que el què arriba al SGBD és una sentència parametritzada, la qual analitza i per a la que elabora el pla d'execució, un sola vegada! Cada vegada que es demana l'execució, el SGBD rep el valor dels paràmetres, que incorpora al pla d'execució i executa. **Estalvi bestial de feina pel SGBD!**

- Evitar injecció de codi (hi ha molts vídeos a la web que ho expliquen)

La injecció de codi només pot passar quan construïm sentències SQL (com en el cas anterior) via concatenació de text i valors de variables cadena, les quals han estat emplenades per un usuari amb vocació de hacker.

➤ Cas de la sentència (1) anterior on el camp `id_dept` és numèric però la variable `dept` és cadena:

Imaginem que un programa demana per pantalla que l'usuari introdueixi el codi de departament i que el seu valor va directament a variable `dept`. Si l'usuari introdueix 10, 20, 30..., cap problema de seguretat doncs en aquest cas (1) executa les consultes següents:

```
select nom, sou from emp where id_dept = 10;
select nom, sou from emp where id_dept = 20;
select nom, sou from emp where id_dept = 30;
```

Però... I si l'usuari és hacker i introdueix, per exemple: `10 or 1=1`

fixeu-vos que en construir la sentència (1) queda:

```
select nom, sou from emp where id_dept = 10 or 1=1
```

i en conseqüència, accedeix a les dades dels usuaris de tots els departaments!!! **Ha injectat codi!!!**

Amb una sentència parametritzada (a més de ser més eficient), això no passaria:

```
s = "select nom, sou from emp where id_dept = ?";
PreparedStatement ps = connection.prepareStatement(s);
ps.setByte(1,dept);
```

En cas que l'usuari hagi introduït `10 or 1=1`, es produirà una excepció en executar `setByte` i no es produirà injecció de codi.

➤ Cas de la sentència (2) anterior on el camp `id_dept` és cadena i la variable `dept` és cadena:

Imaginem que un programa demana per pantalla que l'usuari introdueixi el codi de departament i que el seu valor va directament a variable `dept`. Si l'usuari introdueix 10, 20, 30..., cap problema de seguretat doncs en aquest cas (1) executa les consultes següents:

```
select nom, sou from emp where id_dept = '10';
select nom, sou from emp where id_dept = '20';
select nom, sou from emp where id_dept = '30';
```

Però... I si l'usuari és hacker i introdueix, per exemple: `10' or '1'='1`

fixeu-vos que en construir la sentència (2) queda:

```
select nom, sou from emp where id_dept = '10' or '1'='1'
```

i en conseqüència, accedeix a les dades dels usuaris de tots els departaments!!! **Ha injectat codi!!!**

Amb una sentència parametritzada (a més de ser més eficient), això no passaria:

```
s = "select nom, sou from emp where id_dept = ?";
PreparedStatement ps = connection.prepareStatement(s);
ps.setString(1,dept);
```




En executar la sentència, si l'usuari ha introduït `10' or '1'='1` a la variable `dept`, el SGBD rep aquest valor i l'incorpora com a `String` a la instrucció SQL (canviant `'` per `''`) i executarà:

```
select nom, sou from emp where id_dept = '10'' or ''1'=''1'
```

sense obtenir cap resultat. **Hem evitat la injecció de codi!!!**

i en conseqüència, accedeix a les dades dels usuaris de tots els departaments!!! **Ha injectat codi!!!**

- Projecte 241010_1_ProvaInjeccioCodi: Usa una consulta amb dada a introduir per l'usuari i si l'usuari escriu `10 or 1=1`, mostra la informació de tots els departaments. Injecció de codi! Executeu Prova...
- Projecte 241010_2_ProvaConsultaParametritzada: Projecte anterior retocat amb consulta parametritzada. Executeu Prova...

Procediment per usar consultes parametritzades

1. Es declara un `PreparedStatement`, passant-li la consulta on els paràmetres són `?`. N'hi pot haver varis.
`PreparedStatement ps = connection.prepareStatement(consulta);`
Java envia la sentència al SGBD. Això es fa una única vegada.
2. Cada vegada que es vol executar la consulta, es dona valors a tots els paràmetres, referint-nos-hi per la posició que ocupen dins la consulta, començant per 1:
`ps.setTipusDadaAdequat(numParàmetre, valor);`
3. Una vegada tots els paràmetres han estat emplenats, executem la consulta:
`RecordSet rs = ps.executeQuery();`

Exercici per lliurar a Classroom (tasca 4):

Refer el projecte de la tasca 3 usant sentència parametritzada allà on calgui.

Instruccions NO SELECT (DML-DDL-DCL) via JDBC

15/10/24

- **Recordem** que per executar instruccions `SELECT`, hem d'utilitzar el mètode `executeQuery`:

- Per a consultes no parametritzades:

```
Statement st = Connection.createStatement();
ResultSet rs = st.executeQuery(consultaSelect);
```

- Per a consultes parametritzades:

```
PreparedStatement ps = Connection.prepareStatement(consultaAmbParàmetres);
```

Abans d'executar:

- Emplenem els paràmetres amb mètodes `set`
- Els paràmetres s'indiquen per posició a partir de 1.

Per executar: `ResultSet rs = ps.executeQuery();`

En els dos casos, el resultat és un `ResultSet` que hem de processar.

- **Les instruccions SQL NO SELECT (DML-DDL-DCL) s'executen amb mètodes `executeUpdate`.**

- Per a instruccions NO SELECT no parametritzades:

```
Statement st = Connection.createStatement();
int i = st.executeUpdate(instrucció)
```

- Per a instruccions parametritzades:

```
PreparedStatement ps = Connection.prepareStatement(instruccióAmbParàmetres);
```

Abans d'executar:

- Emplenem els paràmetres amb mètodes `set`
- Els paràmetres s'indiquen per posició a partir de 1.



Per executar: `int i = ps.executeUpdate();`

En els dos casos, el resultat és un `int` que ens indica:

- El número de files processades en cas d'instrucció DML (`INSERT – DELETE – UPDATE`)
- Zero si es tracta d'una instrucció DDL (`create/alter table/index...`) o DCL (`create/drop user, grant, revoke`)

Atenció amb la informació que JDBC dona respecte commit/rollback:

It is strongly recommended that an application explicitly commits or rolls back an active transaction prior to calling the close method. If the close method is called and there is an active transaction, the results are implementation-defined.

És a dir, si intentem tancar una connexió amb canvis pendents de fer `commit/rollback`, JDBC no assegura què succeirà; ho deixa a decisió del fabricant del connector, que pot decidir:

- Generar excepció
- Fer `rollback` automàtic (sembla que seria el més normal)
- Fer `commit` automàtic
- Oracle, en la documentació del seu connector `ojdbc8.jar`, diu:
If the auto-commit mode is disabled and you close the connection without explicitly committing or rolling back your last changes, then an implicit COMMIT operation is run.
- [SQLServer](#), en la documentació diu:
Calling the close method in the middle of a transaction causes the transaction to be rolled back.
- PostgreSQL & MySQL, via comprovació (no trobat en documentació), executen `rollback` en tancar connexió

Davant el fet que no tots els SGBD actuen de la mateixa manera, és altament recomanable, per no dir imprescindible que abans de tancar una connexió, efectuar `rollback`.

Exercici per dimarts, 22 d'octubre (tasca 5)

Desenvolupar programa que afegixi a la taula `PRODUCTE` la columna `DARRER_PVP` i que l'empleni, per a tots els productes de la taula, amb el darrer preu de venda. Aquesta columna ha de permetre valors nuls, doncs és possible que un producte encara no hagi tingut cap venda.

El programa ha d'informar de qualsevol problema que sorgeixi. Així, per exemple, si executeu el programa quan la columna ja existeix, ha d'informar d'aquest fet i avortar l'execució. És clar, que per provar el programa diverses vegades, haureu d'eliminar la columna des d'una connexió a la BD via *SQLDeveloper* o *SQL*Plus*.

Pautes per la solució – Desenvolupar les dues versions següents:

- **Versió 1:**
 1. Afegix la darrera columna amb un `executeUpdate` d'instrucció no parametritzada.
 2. Recupera tots els productes en `ResultSet` amb un `executeQuery` de `SELECT` no parametritzada
 3. Per cada producte calcula el darrer preu de venda amb un `executeQuery` de `SELECT` parametritzada. Cal tenir en compte que aquesta `SELECT` pot retornar una fila (si el producte s'ha venut alguna vegada) o cap fila (si el producte no s'ha venut mai).
 1. Si el producte s'havia venut alguna vegada, actualitza el darrer preu de venda amb un `executeUpdate` d'instrucció parametritzada.
 2. Si el producte no s'havia venut mai, no omple res i la nova columna quedarà amb `NULL`.
- **Versió 2**, on després d'afegir la columna, amb una única instrucció `update`, modifica tots els productes. Evidentment aquesta segona versió és més eficient, doncs amb una única instrucció informem al SGBD què ha de fer per aconseguir emplenar la nova columna.
No sempre serà possible assolir l'objectiu amb una única instrucció, però si ho és, per què usar-ne més?



Aprofundiment en Statement/PreparedStatement sota mateixa/diferent connexió		24/10/24				
Exemples de l'efecte d'utilitzar diferents Statement/PreparedStatement amb una mateixa connexió i amb diferents connexions.						
Projecte 241024_1_JDBC_Statement						
<ul style="list-style-type: none">Programa P01 – 2 Statement en mateixa connexió: Ambdós Statement comparteixen les dades sense necessitat de fer commit per a que cada Statement vegi el què ha fet l'altre... Tenim st1 i st2. Fem un INSERT (empleat 777-Gotera) via st1 i posteriorment recuperem el contingut de la taula utilitzant st1 i també amb st2 i en les dues recuperacions observem la fila inserida.Programa P02 – Statement en diferents connexions. En aquest cas es demostra que no comparteixen les dades. Cal fer commit per a que un vegi les modificacions efectuades per l'altre. Tenim st1 i st2. Fem un INSERT (empleat 8888-Pepeillo) via st1 i posteriorment recuperem el contingut de la taula utilitzant st1 i també amb st2 i només observem la fila inserida en la recuperació efectuada via st1. Si voleu, podeu retocar el programa afegint un con1.commit() abans d'executar la recuperació de les files via st2 i llavors ja hi observareu la fila inserida via st1.						
Els dos programes mostren l'efecte sobre Statement, però el mateix efecte té lloc en PreparedStatement						
Aprofundiment en gestió de ResultSet: Modificar files i tirar enrere		24/10/24				
Fins ara, hem utilitzat els ResultSet per recollir resultats d'instruccions SELECT i els hem recorregut cap endavant. Aquesta és la utilització més bàsica, però ResultSet permet més accions.						
Si es vol disposar de ResultSet més “funcionals”, s'ha d'indicar en el moment de crear l'Statement o el PreparedStatement sobre el què s'executarà la consulta i s'obtindrà el ResultSet.						
<ul style="list-style-type: none">Hi ha un mètode createStatement que permet incorporar dos paràmetres molt interessants: Statement createStatement(int resultSetType, int resultSetConcurrency) throws SQLException on: resultSetType pot valer: ResultSet.TYPE_FORWARD_ONLY (només endavant – per defecte) ResultSet.TYPE_SCROLL_INSENSITIVE (endavant i enrere) ResultSet.TYPE_SCROLL_SENSITIVE (endavant i enrere) resultSetConcurrency pot valer: ResultSet.CONCUR_READ_ONLY (no es pot modificar les files recorregudes – per defecte) ResultSet.CONCUR_UPDATABLE (permet modificar les files recorregudes, eliminar i inserir)Hi ha un mètode prepareStatement que permet incorporar els mateixos paràmetres del mètode anterior: PreparedStatement prepareStatement(String sql, int resultSetType, int resultSetConcurrency) throws SQLExceptionDiferència entre TYPE_SCROLL_SENSITIVE i TYPE_SCROLL_INSENSITIVE:						
Table 17-1 Visibility of Internal and External Changes for Oracle JDBC						
Result Set Type	Can See Internal DELETE?	Can See Internal UPDATE?	Can See Internal INSERT?	Can See External DELETE?	Can See External UPDATE?	Can See External INSERT?
forward-only	no	yes	no	no	no	no
scroll-sensitive	yes	yes	no	no	yes	no
scroll-insensitive	yes	yes	no	no	no	no



Atenció! Per a que un `ResultSet` sigui modificable, cal haver:

- Creat el `Statement` o el `PreparedStatement` adequadament (dient-li `CONCUR_UPDATABLE`) i, a més,
- NO es pot fer "select * " (cal indicar el nom de les columnes a recuperar)
- Només sobre una taula

Més informació sobre el connector JDBC d'Oracle: [Documentació d'Oracle 18c](#) – [Documentació d'Oracle 21c](#)

Projecte exemple: 241024_2_JDBC_ResultSet_CRUD, que mostra:

- Com modificar camps de la fila on està aturat el cursor que recórrer el `ResultSet`.
 - `rs.updateTipus(nomColumna, nouValor)`, per modificar el contingut d'una columna
 - Quan ja s'han fet totes les modificacions dels camps de la fila, cal executar `rs.updateRow()`, moment en el que el programa envia la instrucció `update` de les columnes afectades a la BD. Evidentment, si s'infringeix alguna restricció PK, FK, CK, UN o NN, es produirà una `SQLException`.
El BUCLE1 dins el programa exemple mostra com es modifica el nom de tots els departaments
- Com s'afegeix una fila (cal "moure" el cursor a una fila especial per inserir, de manera similar a quan utilitzem una interfície gràfica d'accés a BD en la que per afegir una fila ens hem de situar a la darrera fila de la graella mostrada per pantalla)
 - `rs.moveToInsertRow()`, per "crear" una nova fila en el lloc on està ubicat el cursor que recorre el `ResultSet`.
 - `rs.updateTipus(nomColumna, nouValor)`, per anar emplenant les columnes
 - Quan ja s'han emplenat tots els camps de la fila, cal executar `rs.insertRow()`, moment en el que el programa envia la instrucció `insert` a la BD. Evidentment, si s'infringeix alguna restricció PK, FK, CK, UN o NN, es produirà una `SQLException`.
El BUCLE2 dins el programa exemple afegeix departaments (del 61 al 63)
- Com eliminar la fila on està aturat el cursor que recorre el `ResultSet`.
 - `rs.deleteRow()` és la instrucció que elimina la fila on està ubicat el cursor, moment en el que el programa envia instrucció `delete` a la BD. Evidentment, si s'infringeix alguna restricció PK, FK, CK, UN o NN, es produirà una `SQLException`.
En el PUNT3 del programa exemple, s'elimina el departament 40 (que no té empleats i per això es pot eliminar i no es genera cap `SQLException`).
- Com tornar a recórrer el `ResultSet`, passant a l'inici amb el mètode `rs.beforeFirst()`.
- També hi ha mètodes `first()`, `last()`, `previous()`, `rowRefresh()`. Mirar la documentació de `ResultSet`.

Atenció! El programa modifica noms, afegeix departaments i elimina un departament i fa `commit...` Per tant, la BD queda modificada i si el torneu a executar, us petarà la inserció (per PK). Haureu de tornar a executar el guió que deixa les taules amb les dades inicials.

Exercici per dimarts, 5/11 (tasca 6) – Es disposa de la classe de dijous 31, per resoldre dubtes al respecte

Desenvolupar programa que, en iniciar, es connecti a la BD on hi hagi les taules de l'esquema `SANITAT` (teniu el guió a l'aula de Classroom), carregui tots els malalts en un `ResultSet` i mostri un menú (per consola) amb opcions:

1. Mostrar malalts => Ha d'invocar mètode `mostrarMalalts()`.
2. Cercar malalt per inscripció => Ha d'invocar mètode `cercarMalaltPerInscripcio()`
3. Cercar malalts per part de cognom => Ha d'invocar mètode `cercarMalaltsPerPartDeCognom()`
4. Refrescar dades => Ha d'invocar mètode `refrescarResultSet()`
0. Sortir

A continuació es detalla el funcionament que ha de tenir cada mètode.

Si creieu que algun(s) mètode(s) ha(n) de tenir algun(s) paràmetre(s), incorporeu-los.

Si creieu que és interessant afegir algun altre mètode adequat per ser invocat des de diversos mètodes, afegiu-lo.



- **Mètode `mostrarMalalts()`**
Ha de fer un recorregut pel `ResultSet` i mostrar els malalts existents, en format:
INSCRIPCIÓ - COGNOM - DOMICILI - DATA NAIX. - SEXE - NSS
amb columnes ben justificades i on la data de naixement es mostri en format `dd/mm/yyyy`.
Si no hi ha cap malalt, cal informar.
En finalitzar, es torna al menú inicial.
- **Mètode `cercarMalaltPerInscripcio()`**
Ha de demanar per consola un codi de malalt.
El mètode ha de controlar que sigui un codi vàlid.
Si no és codi vàlid, s'informa i es surt del mètode.
Si és codi vàlid, es cercarà dins el `ResultSet`.
Si no existeix, s'informa i es surt del mètode.
Si existeix, cal mostrar-lo en format:
INSCRIPCIÓ - COGNOM - DOMICILI - DATA NAIX. - SEXE - NSS
amb columnes ben justificades i on la data de naixement es mostri en format `dd/mm/yyyy`.
En finalitzar, es torna al menú inicial.
- **Mètode `cercarMalaltsPerPartDeCognom()`**
Ha de demanar per consola part de cognom de malalt.
El mètode ha de controlar que sigui valor vàlid.
Si no és valor vàlid, s'informa i es surt del mètode.
Si és valor vàlid, es cercarà dins el `ResultSet` els malalts que el seu cognom contingui la part introduïda per l'usuari, insensible a majúscules-minúscules.
Si no existeix, s'informa i es surt del mètode.
Si es troba (poden ser varis malalts), cal mostrar-los en format:
INSCRIPCIÓ - COGNOM - DOMICILI - DATA NAIX. - SEXE - NSS
amb columnes ben justificades i on la data de naixement es mostri en format `dd/mm/yyyy`.
En finalitzar, es torna al menú inicial.
- **Mètode `refrescarResultSet()`**
Ha de tornar a carregar les dades dels malalts, per si hi ha hagut alguna modificació a la BD.

Observacions:

- La càrrega inicial del `ResultSet` ha de fer exactament el mateix que el mètode `refrescarResultSet`. Per tant, una vegada establerta la connexió, invoquem `refrescarResultSet` per aconseguir la primera càrrega.
- Degut a que el `ResultSet` s'ha de recórrer varies vegades, necessitem poder-nos ubicar al principi. Per aconseguir-ho, cal declarar l'`Statement` a partir del qual s'executa la consulta que obté el `ResultSet`, com:

```
con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,  
                    ResultSet.CONCUR_READ_ONLY);
```


El segon paràmetre el podem posar de "només lectura" per què en el programa demanat, no hem de fer cap modificació a través del `ResultSet`.
- És adequat crear un objecte `SimpleDateFormat` global amb el format que se'ns demana per les dates, per usar-lo allà on faci falta i no haver-lo de crear cada vegada.
- El `ResultSet` es pot crear amb els malalts ordenats per inscripció (no es demana) però així, en `cercarMalaltPerInscripcio` podem aprofitar que el `ResultSet` està ordenat per inscripció per fer una cerca seqüencial en dades ordenades, ja que quan passem del codi d'inscripció que estem cercant, no cal continuar, doncs ja es té la seguretat de que no existirà la inscripció cercada.
- En `cercarMalaltPerPartDeCognom` no ens podem aprofitar de que el `ResultSet` està ordenat per inscripció.

Exercici per dimarts, 12/11 (tasca 7) – Es disposa de la classe de dijous 7, per resoldre dubtes al respecte

Ampliar programa anterior efectuant altes, baixes i modificacions de malalts via el `ResultSet`. Concretament:

5. Inserir malalt => Ha d'invocar mètode `inserirMalalt()`.
6. Eliminar malalt per inscripció => Ha d'invocar mètode `eliminarMalaltPerInscripcio()`
7. Eliminar malalts per part de cognom => Ha d'invocar mètode `eliminarMalaltsPerPartDeCognom()`
8. Modificar malalt => Ha d'invocar mètode `modificarMalalt()`
9. Validar els canvis => Ha de fer `commit`



10. Desfer els canvis => Ha de fer `rollback`

A continuació es detalla el funcionament que ha de tenir cada mètode.

- Mètode `inserirMalalt()`:
Ha de demanar totes les dades d'un malalt.
Només ha de comprovar que les dades són correctes per cada camp, però no cal que comprovi restriccions de PK, FK, CK, UN i NN.
Cal efectuar la inserció via el `ResultSet`, informant si s'ha pogut efectuar la inserció a la BD o si pel contrari s'ha violat alguna restricció, informant en aquest cas del problema (missatge que retorna Oracle).
- Mètode `eliminarMalaltPerInscripcio()`
Ha de fer un tractament similar al `cercarMalaltPerInscripcio()`, però tenint en compte que, si no existeix en el `ResultSet`, s'informa de la no existència i si existeix, cal eliminar-lo (via `ResultSet`), informant si s'ha pogut eliminar de la BD o si pel contrari s'ha violat alguna restricció, informant en aquest cas del problema (missatge que retorna Oracle).
- Mètode `eliminarMalaltsPerPartDeCognom()`
Ha de fer un tractament similar al `cercarMalaltsPerPartDeCognom()`, però tenint en compte que, si no existeix cap malalt en el `ResultSet`, s'informa de la no existència i si existeixen, cal eliminar-los (via `ResultSet`), informant si s'han pogut eliminar de la BD o si pel contrari s'ha violat alguna restricció, informant en aquest cas del problema (missatge que retorna Oracle).
- Mètode `modificarMalalt()`:
Ha de demanar un número d'inscripció de malalt. L'ha de cercar.
Si no existeix, s'informa.
Si existeix, ha de mostrar submenú:
 1. Cognom: <cognom>
 2. Domicili: <domicili>
 3. Data Naix: <data naixement>
 4. Sexe: <sexe>
 5. NSS: <nss>
 0. Finalitzarde manera que l'usuari pugui modificar les dades del(s) camp(s) que vulgui tantes vegades com vulgui
En finalitzar, cal traspasar els canvis a la BD (via `ResultSet`), informant si s'ha pogut efectuar la modificació a la BD o si pel contrari s'ha violat alguna restricció, informant en aquest cas del problema (missatge que retorna Oracle).

Observacions:

- Per poder modificar dades de la BD via un `ResultSet`, aquest no pot ser de només lectura, com havíem fet en versió anterior. Ens cal retocar l'`Statement` a partir del qual s'obté el `ResultSet`, declarant-lo:

```
con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                    ResultSet.CONCUR_UPDATABLE);
```
- Quan tingueu desenvolupat el mètode `inserirMalalt`, comproveu els següents funcionaments:
 - Posar en marxa el programa.
 - Sense *Mostrar Malalts*, executeu una inserció de malalt amb dades que no infringeixin cap de les restriccions pels diversos camps.
 - Posteriorment executeu *Mirar Malalts*. Veureu que el nou malalt està en primera posició. Lògic, doncs no ens havíem mogut pel `ResultSet`.
 - Executeu una nova inserció de malalt, que no provoqui cap error.
 - Torneu a executar *Mirar Malalts*. Veureu que el nou malalt està en darrera posició. Lògica, doncs en el primer *Mirar Malalts*, havíem fet un recorregut i ens havíem quedat al final de tots. Queda clar que `rs.moveToInsertRow` insereix la nova fila en la posició on és el cursor que recorre el `ResultSet`.
 - Si comproveu el contingut de la taula `MALALTS` des de `SQLDeveloper`, no apareix cap dels 2 nous malalts, doncs encara no s'ha validat els canvis.
 - Executeu opció "commit" i comproveu que dins la taula `MALALTS` des de `SQLDeveloper` ja apareixen els nous malalts.



- Crear un malalt amb un codi d'inscripció ja existent. Observeu que, en el moment d'executar `insertRow` és quan "peta", informant que s'ha infringit la restricció única (`MALALT_PK`)

Exercici per dimarts, 19 de novembre (tasca 8)

Desenvolupar programa pensat per intentar eliminar clients, amb el següent funcionament repetitiu

- Demani a l'usuari per consola un codi de client, amb missatge:
Introdueixi codi de client (zero per finalitzar el programa)
El programa ha d'obligar que el valor introduït per l'usuari no sigui negatiu
- Si el valor és zero, finalitza el programa.
- Si el valor no és zero (serà estrictament positiu), ha d'invocar mètode `eliminarClient` encarregat de cercar i eliminar el client indicat.

Respecte el mètode `eliminarClient` ha de controlar totes les possibilitats.

- Que el codi de client sigui vàlid (estrictament positiu), cosa que en el nostre cas no passarà mai, doncs es suposa que el programa ja ho controla, però ho programem per si algun dia usem el mètode en un altre programa.
Si no és vàlid, ha de generar una `RuntimeException` amb missatge adequat.
- Que el client a eliminar existeixi.
Si no existeix, ha de generar una `RuntimeException` amb missatge adequat.
- Que el client es pugui eliminar.
Si no es pot eliminar, ha de generar una `RuntimeException` amb missatge adequat.

El mètode no ha de mantenir cap interacció amb l'usuari. O tot va bé, o si hi ha algun problema ho comunicarà via `RuntimeException`.

El programa principal, en invocar el mètode `eliminarClient`, ha d'estar preparat per esbrinar si tot ha anat bé o si s'ha produït algun problema i informarà a l'usuari de la correcta eliminació o del problema sorgit.

Observació:

És molt probable que en el desenvolupament del mètode `eliminarClient` penseu en executar dues instruccions SQL:

- 1a. Instrucció `select` per esbrinar si el client existeix.
- 2a. Instrucció `delete` per intentar eliminar el client, dins un `try {...} catch (SQLException ex)` per controlar el possible error per violació de FK.

Però si ho penseu millor, fixeu-vos que us podeu estalviar la primera `select`, doncs en executar la instrucció `delete`, podeu comprovar el valor que retorna (número de files eliminades), de manera que un zero indica que no existia el client. Cal sempre cercar el camí més eficient!

Matisse – Creació de BD – Definició de classes – Gestió d'objectes via SQL

14/11/24...

Veure document `DAMDAW_M03_UF6_IntroduccióAlsSGBDOO.pdf` adjunt

[Matisse](#) és un SGBDOO i en [aquesta carpeta](#) es pot trobar les darreres versions 32/64 bits per a Windows. A l'apartat *documentation* de la seva web hi ha informació per procedir a la instal·lació així com guies d'utilització i d'accés des de diferents llenguatges de programació. Nosaltres usarem un escriptori Isard ja preparat.

Eines bàsiques a conèixer (2h)

- *Matisse Enterprise Manager*, interfície gràfica per interactuar amb les BD del SGBD, equivalent a *SQLDeveloper* d'Oracle o *pgAdmin* de PostgreSQL. Funcionalitats a conèixer:
 - ✓ Engregar-aturar-crear-eliminar BD
 - ✓ Crear/actualitzar classes dins BD a partir d'import de fitxer ODL (documentació a [Matisse ODL Programmer's Guide](#)).
 - ✓ Usar *SQLAnalyzer* per accedir a la BD i consultar/inserir/modificar/eliminar objectes amb el llenguatge SQL que facilita Matisse (documentació a [Matisse SQL Language Reference](#)). Matisse no facilita OQL.
- *Database Modeler*, eina per generar UML a partir de BD engegada i per crear/actualitzar classes a la BD des del disseny UML. Alerta... la modificació de classes des d'aquesta eina presenta anomalies...



La instal·lació de Matisse facilita dos arxius ODL d'exemple dins `<pathMatisse>\schema\ODL` i dos arxius XML dins `<pathMatisse>\data\XML` amb dades d'exemple per poblar BD creades a partir dels arxius ODL.

Matisse facilita un llenguatge DDL similar a SQL-DDL per crear classes. És "invent" de Matisse.

L'escriptori Isard que es facilita incorpora BD Media creada a partir d'importació del fitxer `MediaSchema.odl` facilitat per Matisse i poblada amb dades a partir del fitxer `media.xml`.

- ✓ Observar la definició de camps (`attribute`), relacions (`relationship`) i índexs (`mt_index`) en el fitxer `odl`. Trobareu sintaxis de cada definició en apartats 2.4-2.5-2.6 de [Matisse ODL Programmer's Guide](#).
- ✓ No hi ha la possibilitat de definir PK en una classe... En BDOO tots els objectes tenen un OID (Object Identifier) automàtic, que no canvia i que no es reutilitza quan l'objecte s'elimina.
- ✓ Per garantir la unicitat d'algun camp o conjunt de camps en els objectes d'una classe, cal crear un índex únic.
- ✓ També hi ha la possibilitat de definir diccionaris (`mt_entry_point_dictionary`) per facilitar l'accés a objectes per valor d'alguns camps (similar a les classes Java que implementen la interfície `Map -TreeMap, HashMap, ...`). No ho utilitzarem.
- ✓ Les classes en Matisse també poden contenir mètodes, funcionalitat que no veurem.
- ✓ Executar instruccions `SELECT, INSERT, DELETE, UPDATE` sobre la BD Media, que sempre podeu tornar a crear i emplenar en cas de destroça.

FYI: En [aquest vídeo](#) hi ha una introducció molt més avançada, per si voleu aprofundir en el tema.

Exercici per dimarts, 26 de novembre (tasca 9)

Crear guió ODL amb dues classes per gestionar dades en Matisse equivalents a les dades existents a les conegudes taules `DEPT` i `EMP` de l'esquema `empresa` de la BD Oracle, tenint en compte:

- Les classes es poden anomenar `departament` i `empleat`.
- Cal definir cada camp amb el tipus de dada equivalent als tipus de dades dels corresponents camps en les taules `DEPT` i `EMP`, i, per les cadenes, amb la llargada equivalent, de manera que en un proper exercici, puguem crear dins Matisse els objectes "equivalents" a les files existents en les taules `DEPT` i `EMP`.
- Cal aconseguir que els codis de departament i d'empleat siguin únics (via índex únic)
- Cal incorporar índexs equivalents als índexs que tenen les taules `DEPT` i `EMP` si és el cas.
- Les relacions `DEPT-EMP` i `EMP-EMP` (cap) han de ser bidireccionals.

Persistència d'objectes en Matisse des de Java

21/11/24

En [aquest vídeo](#) hi ha una introducció a l'accés des de Java a Matisse.

FYI: En [aquest altre](#) hi ha una introducció molt més avançada, per si voleu aprofundir en el tema.

Matisse genera automàticament la capa de persistència per als objectes de les classes definides a la BDOO i per diversos llenguatges: C#, VB.NET, Java, C++, PHP, Python i Eiffel.

Com generar capa de persistència en Java per BD Matisse:

Es pot fer de diverses maneres:

- Des de consola, amb l'eina `mt_sdl` a partir d'un fitxer ODL que contingui la definició de les classes:
`mt_sdl stubgen -lang java fitxer.odl`
- Des de l'*Enterprise Manager*, per l'opció *Schema|Generate Code* (fet a classe)
- Des de *Data Modeler*, per l'opció *Diagram|Generate Code* amb l'esquema UML obert.

Dins el SGBD Matisse, podem tenir les classes organitzades en diversos `namespaces`. Això, en el fitxer ODL s'aconsegueix muntant una estructura com la següent:

```
module nom1 {
  module nom11 {
    interface Class1 ...
    ...
    interface Class2 ...
    ...
  }
};
```




```
module nom12 {  
    interface Class1 ...  
    ...  
    interface ClassB ...  
    ...  
};  
};  
module nom2 {  
    ...  
};
```

L'estructura anterior correspon a la definició, dins la BD de *Matisse*, de les classes:

```
nom1.nom11.Class1  
nom1.nom11.Class2  
nom1.nom12.Class1  
nom1.nom12.ClassB  
nom2....
```

Des de l'*Enterprise Manager (Schema|Generate Code)*, per generar la capa de persistència d'un namespace:

- Indicarem a *Project Directory* la carpeta (ha d'existir) on volem que es generi la capa de persistència.
- Seleccionarem el llenguatge
- Marcarem la casella *Database namespace* i seleccionarem el namespace que correspongui
- Marcarem, si volem que les nostres classes es generin en un determinat paquet, la casella *Language package* i escriurem el nom del paquet on han de pertànyer les classes a generar.
- Seleccionarem la casella *SQL method calls* en cas que es desitgi la generació de mètodes que permetin executar els mètodes SQL enregistrats a la BD. Si s'escull, no podem tenir mètodes amb nom `toString` dins les classes de *Matisse* per què el mètode generat xocaria amb mètode `toString` de les classes Java.

El paquet generat el podem incorporar dins la carpeta `src` d'un projecte NetBeans, el qual haurà de tenir accés al fitxer `matisse.jar` que es pot trobar en el directori `lib` de qualsevol instal·lació de *Matisse* (Windows o Linux) i també [aquí](#) (dins `Matisse9.1.12.Jar.zip`). Interessant crear a NetBeans una llibreria de nom *Matisse* que incorpori aquest `.jar`.

La documentació de l'API Java de *Matisse* es troba a `docs/java/api` de qualsevol instal·lació de *Matisse* (Windows o Linux) i [aquí](#) (dins `Matisse9.1.12JarJavadoc.zip`). Interessant afegir la documentació a la llibreria instal·lada en NetBeans.

Amb la llibreria *Matisse* instal·lada en el projecte NetBeans i el paquet de classes generat dins la carpeta `src` del projecte, podem generar el `javadoc` corresponent a les classes generades.

Requeriments en Java per poder executar programes que usin capa de persistència generada per Matisse:

- Des de Windows: Cal que el S.O. de la màquina on s'executi el programa, que pot ser la mateixa on hi ha el SGBD o no, incorpori en el seu `path` l'accés a les llibreries (`dll`) de *Matisse* per Windows.

Com disposar d'aquestes llibreries?

- o Si el SGBD *Matisse* està a la mateixa màquina, ja les tenim dins `pathMatisse\bin`.
- o Si el SGBD *Matisse* està en una altra màquina:
 - Podem instal·lar el client *Matisse* (no el SGBD) i tindrem les llibreries dins `pathMatisse\bin`.
 - Però no és imprescindible instal·lar el client *Matisse*, doncs n'hi ha prou amb copiar les `dll` dins una carpeta i incorporar la ruta de la carpeta en el `path` del SO.
Atenció! Si es fa així, cal utilitzar les `dll` de la mateixa arquitectura (32 bits/64 bits) existent en la màquina Windows des d'on executem el programa.
Les `dll` necessàries les trobareu [aquí](#) (dins `Matisse9.1.12Bin##bits.zip`).

NetBeans s'ha de reobrir després d'haver instal·lat el client *Matisse* o haver modificat el `path` del SO per a que trobi les `dll` en executar l'aplicació.

- Des de Linux: També és possible. En el Linux client cal haver instal·lat algunes llibreries de *Matisse*, de forma similar a les `dll` de Windows.

Codi Java a tenir en compte per connectar amb BD de Matisse i gestionar les seves dades:

1. Per poder establir connexió amb una BD d'un SGBD *Matisse*, cal utilitzar el constructor:



MtDatabase db = new MtDatabase(màquina, nomBaseDeDades, xxx)
on xxx és un MtObjectFactory que es pot generar des de diverses classes:

- Si les classes a la BD no estan en cap namespace i les classes generades tampoc estan en cap paquet:
`new MtPackageObjectFactory("","");`
- Si les classes a la BD no estan en cap namespace però les classes generades pertanyen a un paquet:
`new MtPackageObjectFactory("paquetJava","");`
o també, utilitzant el fitxer `nomBaseDades_stubsSchemaMap.txt` generat en el procés de creació de les classes, efectuant la crida:
`new MtExplicitObjectFactory("nomBD_stubsSchemaMap.txt");`
- Si les classes a la BD estan en un namespace i les classes generades s'ubiquen en un paquet:
`new MtPackageObjectFactory("paquetJava","namespaceMatisse");`
o també, utilitzant el fitxer de mapatge `nomBaseDades_stubsSchemaMap.txt` generat en el procés de creació de les classes, efectuant la crida:
`new MtExplicitObjectFactory("nomBD_stubsSchemaMap.txt");`

En cas d'utilitzar el fitxer de mapatge entre les classes Java i les classes *Matisse* que genera, es produeixen un seguit d'avisos que semblen deguts a que mapa unes classes inexistents... Es poden comentar i desapareixen els avisos...

L'objecte `MtDatabase` només és una definició i no estableix encara la connexió.

2. Procés a seguir:

- Obrir un canal de comunicació amb el mètode `MtDatabase.open()`, moment en el que s'intenta establir connexió. La BD ha d'existir i ha d'estar oberta.
- Les operacions d'accés a la BD han d'estar dins una transacció... Hi ha dos tipus de transaccions:
 - De només lectura (`VersionAccess`) amb els mètodes `startVersionAccess` i `endVersionAccess` per iniciar i finalitzar la transacció.
 - Per modificació (`Transaction`) amb els mètodes `startTransaction`, `commit` i `rollback`.

3. Recordar de tancar la connexió amb la BD usant mètode `close` sobre la `MtDatabase` oberta.

4. L'API genera `MtException` pels errors que es produeixen i aquesta classe és `RuntimeException`, motiu pel que Java no ens obliga a gestionar-les però cal incorporar tot el codi dins un `try... catch...` per si se'n genera alguna.

El projecte `2411XX_ProgramesMatisseMedia` conté diferents programes que interactuen amb la BD `media`.

- ✓ P01: Estableix connexió amb la BD i finalment tanca la connexió.
- ✓ P02: Programa que llista les persones existents a la BD. Usa `instanceIterator`, que mostra les persones ordenades per `OID`.
- ✓ P03: Programa que obté les persones existents a la BD ordenades per cognom-nom. Això es pot aconseguir directament de la BD per què la classe `Person` disposa d'un índex per cognom-nom (`PersonName`) i, en conseqüència, la classe Java generada disposa de mètodes `personNameIterator` per recuperar els objectes segons l'ordre marcat per l'índex.
- ✓ P04: Programa que cerca una pel·lícula per nom introduït per usuari i mostra títol i director. Cal tenir present que una pel·lícula pot no tenir director i cal controlar-ho. Dues possibilitats:
 - v1: Usant l'índex `movieNameIdx` (mètode `movieNameIdxIterator`) que incorpora la classe `Movie`.
 - v2: Usant mètode `lookupObjectsMovieNameIdx`, que permet localitzar tots els objectes amb un nom concret, via índex `movieNameIdx` de la classe `Movie`. Cal usar aquest mètode per què el nom de la pel·lícula no és únic. Existeix mètode `lookupMovieNameIdx` per trobar algun objecte amb el nom indicat.



- ✓ P05: (Exercici per dijous 28 de novembre – tasca 10) Programa que mostri la fitxa dels socis (`Membership`) ordenats per codi (`memberID`). Per cada soci cal mostrar:

- Codi
- Cognom, Nom
- Data d'alta
- Número de telèfon
- Pel·lícules que ha prestat: títol, llargada, categoria.

Useu l'índex `memberIDidx` de la classe `Membership`.

En els següents programes practicareu el DML. En els programes Java que fem usant JDBC per gestionar la persistència d'objectes dins la BD, ens veiem obligats a invocar sentències SQL:

- `insert` per enregistrar dins una taula a la BD, un objecte que tinguem en memòria i que encara no havíem afegit a la taula.
- `update` per enregistrar canvis en un objecte que tenim en memòria i que ja teníem en una taula a la BD.
- `delete` per eliminar la fila corresponent a un objecte que puguem tenir en memòria.

És a dir:

- quan fem la creació d'un objecte en memòria, no és automàtic la seva incorporació a la taula.
- quan modifiquem algun camp en un objecte en memòria, el canvi no és automàtic a la taula.
- quan destruïm un objecte en memòria, l'eliminació no és automàtica a la taula.

En BDOO no hi ha distinció entre objecte en memòria i objecte dins la BD. Són el mateix!!! Practiquem-ho!!!

- ✓ P06: Programa que permeti afegir una persona (nom i cognoms entrats per l'usuari sense comprovar si ja existeix una persona amb mateix cognom-nom), indicant si és només persona o si és artista o director.

Cal obrir una `Transaction` (no una `VersionAccess`) per què anem a executar instruccions DML.

Cal usar el constructor que pertoqui (`Person` o `Artist` o `MovieDirector`).

Observar que els constructors que facilita Matisse no tenen atributs (inicialment tots estan a `null`).

Caldrà emplenar els camps que correspongui amb els corresponents mètodes `setter`.

Per enregistrar els canvis a la BD cal fer `commit`.

Podeu comprovar que si s'intenta fer un `commit` i algun camp obligatori no està ple, apareix excepció.

Abans de tancar, si hi ha alguna `Transaction` activa, farem `rollback` (mai `commit` si programador no ho explicita).

Com que la classe `Person` no té un índex únic per la parella cognom-nom, Matisse permet tenir diferents persones amb mateix cognom-nom.

Els canvis efectuats els podem comprovar a la BD via `SQLAnalyzer`.

- ✓ P07: Programa que permeti canviar el cognom a les persones que tenen un determinat cognom. L'usuari ha d'introduir cognom a canviar i nou cognom.

En tenir la sort de disposar a `Person`, d'un índex per cognom, podem demanar un iterador que doni directament les persones que tenen el cognom indicat i procedir a canviar-los el cognom, invocant el corresponent mètode `set`. Caldrà un `commit`.

Els canvis efectuats els podem comprovar a la BD via `SQLAnalyzer`.

- ✓ P08: Programa que permeti eliminar les persones que tenen determinats cognom i nom introduïts per l'usuari.

Cercarem les persones i les eliminarem amb el mètode `remove`. Caldrà un `commit`. Dues possibilitats:

V1: Usant l'índex `personName` (mètode `personNameIterator`) que incorpora la classe `Person`.

V2: Usant mètode `lookupObjectsPersonName`, que permet localitzar tots els objectes amb un cognom i nom concrets.

- ✓ P09: (Exercici per dimarts 3 de desembre – tasca 11) Programa que permeti crear un soci seguint els passos:

- L'usuari ha de demanar identificador del nou soci
- Ha de comprovar que no existeix cap soci amb aquest identificador. Si existeix, informa i finalitza.
- Si no existeix, demana totes les dades necessàries per poder-lo enregistrar amb telèfon no buit.
- Finalment, s'ha d'introduir que aquest soci ha prestat totes les pel·lícules de la categoria `Comedy`.



PRÀCTICA – Tasca 12 – Puntua 20% en nota mòdul amb nota mínima 5

Desenvolupar projecte `T12_Cognom1Cognom2Nom`, amb els següents programes, que cal executar des de màquina física amb túnel Wireguard obert contra escriptoris que contenen Oracle i Matisse.

El driver per Oracle ha de residir en llibreria del NetBeans anomenada `JDBC Oracle`.

El driver per Matisse ha de residir en llibreria del NetBeans anomenada `Matisse`.

Les dades necessàries per connectar:

- amb l'esquema Oracle que s'indiqui han de residir en fitxer de configuració `Cognom1-Oracle.properties`
- amb la BD Matisse que s'indiqui han de residir en fitxer de configuració `Cognom1-Matisse.properties`

En Oracle, sempre que tingui sentit, usar `PreparedStatement` enlloc de `Statement` per guanyar eficiència i evitar injecció de codi.

Incorporeu en l'arrel del projecte, el guió que vàreu lliurar en la tasca 9. En Matisse, cal tenir BD de nom `Empresa` amb classes generades pel aquest guió. Si en desenvolupar la pràctica us adoneu que teniu algun error en el guió que vàreu lliurar en tasca 9, arregleu-ho. El projecte ha de contenir el guió definitiu.

1. Programa `P01_MigracióOracleToMatisse` dins paquet `org.milaifontanals.m03uf6`:
 - Intenti connectar amb l'esquema `empresa` del servidor Oracle i amb una BD de nom `Empresa` creada dins el SGBD Matisse amb el guió creat en tasca 9. Si alguna connexió no s'assoleix, avisa i finalitza.
 - Traspassis els departaments i els empleats de l'esquema `empresa` d'Oracle dins la BD `Empresa` de Matisse.
2. Programa `P02_InsertEmpleatEnOracle` dins paquet `org.milaifontanals.m03uf6`:
 - Intenti connectar amb l'esquema `empresa` del servidor Oracle. Si no es pot, avisa i finalitza.
 - Demani codi d'empleat a inserir.
 - Comprovi la inexistència. En cas d'existir, informi de l'existència mostrant dades de l'empleat amb nom de corresponent departament i del cap (si en té) i finalitza.
 - Demani codi de departament per l'empleat.
 - Comprovi l'existència. En cas de no existir, informi de la inexistència i finalitza.
 - Demani la resta de dades, obligant valor només per les obligatòries.
 - Insereixi l'empleat a la BD, informant de la correcta inserció o de la causa d'error en cas contrari.
3. Programa `P03_InsertEmpleatEnMatisse` dins paquet `org.milaifontanals.m03uf6`:
 - Intenti connectar amb la BD `Empresa` del servidor Matisse. Si no es pot, avisa i finalitza.
 - Demani codi d'empleat a inserir.
 - Comprovi la inexistència. En cas d'existir, informi de l'existència mostrant dades de l'empleat amb nom de corresponent departament i del cap (si en té) i finalitza.
 - Demani codi de departament per l'empleat.
 - Comprovi l'existència. En cas de no existir, informi de la inexistència i finalitza.
 - Demani la resta de dades, obligant valor només per les obligatòries.
 - Insereixi l'empleat a la BD, informant de la correcta inserció o de la causa d'error en cas contrari.
4. Programa `P04_EliminarEmpleatEnOracle` dins paquet `org.milaifontanals.m03uf6`:
 - Intenti connectar amb l'esquema `empresa` del servidor Oracle. Si no es pot, avisa i finalitza.
 - Demani codi d'empleat a eliminar.
 - Comprovi l'existència. En cas de no existir, informi de la inexistència i finalitza.
 - Si existeix, comprovi si és cap d'algun empleat. Si ho és, informi i demani confirmació per prosseguir amb eliminació, deixant els seus subordinats sense cap.
 - Informant de la correcta eliminació i de quants subordinats (si és el cas) s'han quedat sense cap.
5. Programa `P05_EliminarEmpleatEnMatisse` dins paquet `org.milaifontanals.m03uf6`:
 - Intenti connectar amb la BD `Empresa` del servidor Matisse. Si no es pot, avisa i finalitza.
 - Demani codi d'empleat a eliminar.
 - Comprovi l'existència. En cas de no existir, informi de la inexistència i finalitza.
 - Si existeix, comprovi si és cap d'algun empleat. Si ho és, informi i demani confirmació per prosseguir amb eliminació, deixant els seus subordinats sense cap.
 - Informant de la correcta eliminació i de quants subordinats (si és el cas) s'han quedat sense cap.



6. Programa `P06_GestióDepartamentViaResultSet` dins paquet `org.milaifontanals.m03uf6`:
- Intenti connectar amb l'esquema `empresa` del servidor Oracle. Si no es pot, avisa i finalitza.
 - Carregui els departaments en un `ResultSet`
 - Menú amb següents opcions per gestionar departaments, via `ResultSet`:
 0. Finalitzar, que abandoni el programa desfent els canvis no validats.
 1. Alta departament, que permeti inserir un departament amb dades introduïdes per usuari.
Haurà de constatar que l'alta s'ha pogut efectuar o informar d'error en cas contrari.
 2. Baixa departament, que permeti eliminar un departament, amb codi introduït per usuari.
Haurà de constatar que la baixa s'ha pogut efectuar o informar d'error en cas contrari.
 3. Cercar departament, que permeti mostrar un departament, amb codi introduït per usuari.
 4. Modificar departament, que permeti modificar un departament, amb codi introduït per usuari.
Cal poder modificar codi-nom-localitat
Haurà de constatar que la baixa s'ha pogut efectuar o informar d'error en cas contrari.
Recordar que en Oracle, una PK es pot modificar si no és referenciada.
 5. Validar canvis
Abans d'executar aquesta opció, es pot comprovar que els canvis efectuats amb les opcions 1-2-4 no són definitius a la BD, obrint una sessió via `SQLDeveloper` i comprovant contingut de la taula `DEPT`.
 6. Refrescar departaments, que s'encarregui de tornar a carregar el `ResultSet`.